Contents

xsysroot

Swiss army knife to manipulate operating system images

Integrate GNU/Linux systems for single-board computers.

xsysroot is free software - download the pdf version of this page

Albert Casals - 2015, 2016 - *albert@mitako.eu*

Introduction

The micro computers embeded boards market is evolving at a very fast pace.

The introdution of the RaspberryPI, the Beaglebone, Cubieboard and so many others, allows for a broad range of possibilities both in hardware and software.

Most of these boards are capable of running a regular Linux kernel with a full fledged GNU system. Normally these boards boot the operating system from a small storage device like SD cards, micro SD, eMMC, sometimes embedded EEPROMS on the board itself.

Preparing and installing an operating system on these devices has a few subtleties that make the process a bit different.

xsysroot is a tool that tries to minimize the steps needed throughout this process, making it easier to transport the OS to the storage devices, and put them to run on these boards.

Linux based OSes

There exist countless Open Source Linux based operating systems for these embedded boards, many have been adapted from the former official versions running on bigger systems.

Normally these come in downloadable images which contain a complete file system with the boot loader and the operating system files, generally separated in different partitions.

Mounting these images outside the boards is one of the tasks of xsysroot, which allows for modifying the contents more comfortably without needing the board.

The ARM emulation problem

Most times we are still used to work on Intel based systems, whereas the embedded boards are most commonly based on ARM architecture, they consume much less energy providing an increased performance, need less transistors to do the job, and therefore their production cost is cut down.

However this imposes a challenge when we need to build software for them, which is commonly resolved using on-the-fly ARM emulation using QEMU, or by setting up a cross build chain.

xsysroot relies on Qemu to allow the execution of common tasks, and it also provides for cross compilation wrappers that help build software much faster.

This combination allows for generating ARM code from withing a Intel computer, which can be your laptop, desktop, a Virtual Machine or even a remote VPS.

Installing and upgrading xsysroot

xsysroot runs on ARM and Intel architectures running a recent GNU/Linux system.

Requirements

For Intel systems, a Debian OS with the following packages installed:

- qemu-user-static, qemu-utils, binfmt-support

An account with sudo password-less access.

In both Intel and ARM architectures, you will need NBD kernel support. if `modprobe nbd` does not complain, you are ready to go.

Installation

Insert the nbd module in your kernel: `nbd max_part=xx' in your`/etc/modules`file. Set`xx‘ to match the number of concurrent images you want to mount.

It is also possible to install it on a Intel Virtual Machine or a remote VPS. xsysroot is contained in one single Python module, installation is easy:

```
$ sudo curl -L https://raw.githubusercontent.com/skarbat/xsysroot/master/xsysroot
     > /usr/bin/xsysroot
$ sudo chmod +x /usr/bin/xsysroot
```

Upgrades are also straight forward:

```
$ sudo xsysroot -U
Contacting github...
Upgraded your /usr/local/bin/xsysroot version from 1.802 to 1.906
```

What else do I need?

xsysroot relies on quite a few external tools. Some of them are needed, while some others are optional, depending on the tasks you need to do.

The one core mandatory tool needed is `nbd`. If you can insert this Linux kernel module on your system, you can start using xsysroot. NBD stands for Network Block Device and allows for mounting image files as if they were physical disks.

`xsysroot --tools` will tell you which other tools you might need to install.

Tutorial A: Build Raspbian for the RaspberryPI

The following sections will guide you through the steps of creating a minimal Raspbian image to run on the RaspberryPI.

Throughout the documentation we will refer to the *host* and the *guest* systems. The host is the system where you installed xsysroot, the guest is the image containing Raspbian.

Step 1: Creating a blank image

The first thing we need is a blank image on which to install Raspbian, so let's ask xsysroot to create one. It will have a layout of two partitions: A boot FAT and root ext4. Boot will contain the RaspberryPI firmware and the linux kernel. Root will hold the Raspbian OS, which is basically a Debian GNU system.

```
$ xsysroot --geometry "raspbian.img fat32:40 ext4:800"
creating 840MB image file raspbian.img...
 partition 0 type fat32 size 40MB
 partition 1 type ext4 size 800MB
formatting partitions... done!
```

Step 2: Defining the image

It is time to install Raspbian inside the image's root partition. We need to tell xsysroot how to access the image we just created, this is done using a profile.

Profiles are defined in *json* syntax. Let's create one.

Open an editor to create a file on your home directory called `xsysroot.conf` with the following contents:

```
{
  "raspbian" : {
     "description": "Minimal Raspbian image",
     "nbdev" : "/dev/nbd20",
     "nbdev_part" : "p2",
     "sysroot" : "/tmp/raspbian-root",
     "boot_part" : "p1",
     "sysboot" : "/tmp/raspbian-boot",
     "tmp" : "/tmp",
     "backing_image": "~/raspbian.img",
     "qcow_image": "~/raspbian.qcow"
  }
}
```

Let's look at the special meaning of these fields.

`nbdev` tells xsysroot which NBD device should be used to map the image. Every image needs to have a exclusive NBD device number allocated.

`nbdev_part` and `boot_part` are the partition numbers within the image. This allows xsysroot to expose them on your *host* system through simple directories, specified by `sysroot` and `sysboot`. Recall that we created 2 partitions, boot and root.

If `sysboot` is specified, a special mount point will also be made available from within the image, this allows programs running inside the image to access it painlessly. In the case of Raspbian this is specially important to upgrade the kernel (the so popular rpi-update tool).

`backing_image` refers to the original image file, and `qcow_image` is an incremental version of the first. In practice, this means that xsysroot will never modify the backing image, but all changes will be contained in the qcow image.

Step 3: Accessing the image

Once the xsyroot profile is setup, and the `backing_image` ready, it's time to put the image to work.

The very first time, we need to `--renew` the image. This means that xsysroot will read the `backing_image`, and generate the `qcow_image`. Both images are now said to be *bound*. From here on, we can mount and unmount as many times as we need. Read the Qcow chapter to learn more about how this works.

Let's renew the image:

```
$ xsysroot --profile raspbian --renew
Creating qcow image /home/albert/raspbian.qcow of original size
Formatting '/home/albert/raspbian.qcow', fmt=qcow2 size=840000000
 backing_file='/home/albert/raspbian.img' encryption=off cluster_size=65536
binding qcow image: /home/albert/raspbian.qcow
mounting root partition /dev/nbd20p2 -> /tmp/raspbian-root
mount: mount point /tmp/raspbian-root/dev does not exist
mount: mount point /tmp/raspbian-root/proc does not exist
mount: mount point /tmp/raspbian-root/sys does not exist
mount: mount point /tmp/raspbian-root/tmp does not exist
mounting boot partition /dev/nbd20p1 -> /tmp/raspbian-boot
mount: mount point /tmp/raspbian-root/boot does not exist
Mount done
Preparing sysroot for chroot to function correctly
cp: cannot create regular file `/tmp/raspbian-root/usr/bin': No such file or directory
chroot: failed to run command `/bin/bash': No such file or directory
Preparation done
```

```
Renew done
```

wow… how scary!

Three important things happened. The first is that the qcow image has been generated. This allows us to work on the image without ever affecting the original backing image. It also gives us the option to rollback all changes at any point, by calling `--umount` followed by `--renew` (more on that later).

The second thing is that xsysroot mounted the image to give us access to it from the host. Because the image is empty, xsysroot was not able to map some additional mount points, but that's ok for now.

The third thing is that xsysroot tried to install the ARM emulator inside the image, but because there is not OS yet, it was not possible. Not to worry, time to install Raspbian in it!

Step 4: Installing Raspbian inside the image

We will use the wonderful debootstrap tool. Debootstrap basically takes a network Debian repository as a source, and a directory as an output. On completion, you have a minimal Debian system in it. Let's go!

The debootstrap will be done in 2 stages. The first one is executed from the *host*, and basically pulls all the packages from the network and copies them in the output directory. The second stage is run on the *guest*, which unfolds and installs the packages. Let's do the first stage:

```
$ sudo debootstrap --no-check-gpg --verbose --foreign --variant=minbase
 --arch=armhf jessie $(xsysroot --query sysroot)
 http://mirror.us.leaseweb.net/raspbian/raspbian/
I: Retrieving Release
I: Retrieving Packages
...
```

This process will take a few minutes. Notice the `--query` parameter above, it asks xsysroot to tell us where the root partition can be reached from the host.

Now the second stage needs to be done. For this we will first manually copy the ARM emulator, and call deboostrap in the *guest* to complete it.

We needed to install the emulator manually because we started from an empty image, normally xsysroot would do that for us.

```
$ sudo cp $(which qemu-arm-static) $(xsysroot -q sysroot)/usr/bin
$ xsysroot -x "/debootstrap/debootstrap --second-stage"
```

Great! We now have a minimal Raspbian system installed!

Step 5: Installing Raspberry firmware

At this point Raspbian is installed on the root partition, but we need to install the firmware and the Linux kernel in the boot partition.

This is the first thing the RaspberryPI will look for when you power it up.

Recall from step 3, that some mount points were not available due to the image being empty? Well, now they should be there. Let's unmount and mount the image so that xsysroot finds them.

```
$ xsysroot --umount
$ xsysroot --mount
```

To install the kernel we will use rpi-update. We first need to tell `apt` tool to point to Raspbian, install `curl` and `binutils`, and run `rpi-update` !

We are going to open a ARM shell inside the image to do so.

```
$ xsysroot --chroot
Starting sysroot shell into: /tmp/raspbian-root as the superuser
$ echo "deb http://archive.raspbian.org/raspbian jessie main contrib non-free"
 > /etc/apt/sources.list
$ apt-get update
$ apt-get install curl binutils -y
$ curl -L --output /usr/bin/rpi-update
 https://raw.githubusercontent.com/Hexxeh/rpi-update/master/rpi-update
$ chmod +x /usr/bin/rpi-update
$ /usr/bin/rpi-update
```

The kernel is now installed on the boot partition. We are going to need to tell it where the root partition is located:

```
$ echo "dwc_otg.lpm_enable=0 logo.nologo console=tty1 root=/dev/mmcblk0p2
  rootfstype=ext4 elevator=deadline rootwait quiet" > /boot/cmdline.txt
```

There is one last important bit. The superuser has no password.

```
$ passwd
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

That's it! The image is ready to boot on the RaspberryPI. Press Ctrl-D or type `exit` to escape from the ARM shell.

Step 6: Booting the image

Recall from step 3, that the `qcow_image` holds all the changes applied to the image, and that the `backing_image` is never touched. It still contains the empty image we created on step 1.

That means that Raspbian is actually on the qcow image, so we need to take this one and convert it to a raw format. Let's do that:

```
$ xsysroot --umount
$ qemu-img convert $(xsysroot --query qcow_image) raspbian-bootable.img
```

Ready! We can now burn the image to an SD card and boot it on the RaspberryPI.

```
$ sudo dd if=raspbian-bootable.img of=/dev/sda bs=4M
```

`/dev/sda` would be the physical disk where the SD Card is connected to your computer.

Notice that the image fits in 1GB SD card, and that the root partition still has about 200MB free space. Enjoy!

QCow and backing images

xsysroot makes the images available through QEmu QCOW image format.

The qcow technology allows for a number of advantages, for example recreating the source image any number of times to rollback any changes.

Each time you `--renew` a xsysroot profile, the qcow image will be recreated from scratch, and you will be working with an exact same copy of the original backing image.

The backing image is never written into. The qcow is an incremental list of changes and it always plays along with the backing image. Removing the backing image makes the qcow version completely useless.

This combination allows to rollback changes easy and fast, for example:

```
$ xsysroot --renew
$ touch $(xsysroot -q sysroot)/i_was_here
$ xsysroot --umount
$ xsysroot --renew
$ ls -l $(xsysroot -q sysroot)/i_was_here
file not found
```

Another advantage is that for small changes, the qcow image will be relatively small, it becomes a convenient way to transport changes on the original image across the network by simply sending the qcow image instead.

Once the backing_image has been renewed, you can mount and unmount it as many times as you need, effectively working on the qcow image.

The special `/tmp` directory

It is frequently the case that you need to access files that reside on the host from the guest. To make this simpler, xsysroot will expose a directory of your choice on the host through the `tmp` setting.

Set the `tmp` key in the xsysroot.conf file to point to a directory on your host system. Xsysroot will bind this directory each time you mount it, and release it when you unmount it.

This means that all files will be preserved on your host and reachable from the guest from its `/tmp/` folder.

Xsysroot and security contexts

xsysroot will chroot into the image as root, using the `--chroot` option. It is also convenient to run programs as a different user inside the image, like this

```
$ xsysroot -x "@guest_user bash"
$ whoami
guest_user
```

Another advantage is the use of the sudo tool inside the guest. There is a special option which should make sudo work correctly without prompting for passwords, `--jail`.

In Jail mode, the commands `poweroff`, `shutdown` and `reboot` will be disabled, if found. This allows for a more protected space, which can be specially useful for testing software.

```
$ xsysroot --execute "@user bash"
$ whoami
user
$ sudo whoami
root
```

Networking

Networking from withtin the image is readily available, because xsysroot maps the `/proc` filesystem into the host. The guest will be accessing the outside network from the host IP address.

However, the DNS records can be isolated to point to different name servers. xsysroot by default does it for you, it makes the guests point to `109.69.8.34`. You can change it to your preferred one by changing the Xsysroot script.

Virtual displays

It is also possible to run X11 apps inside the guest.

If you install `Xvfb` on the host, xsysroot will allow the guest to run X11 apps which will be connected to a fake Xserver. Because there is no physical display, you will need to take a screenshot with the `--screenshot` option to see their GUI appearance.

Alternatively, you can install `x11vnc` on the host if you need to interact with X11 GUI apps. In this case, xsysroot will start a vnc server attached to the xsysroot. This will allow you to bring the display to a remote device using a VNC client.

In either case the virtual displays will start and stop each time you mount and umount the image, respectively. When opening a shell inside the guest with `--chroot`, xsysroot will automatically export the `DISPLAY` to point to the virtual display.

Each xsysroot profile has its own independent virtual display, which means you can have one virtual display per xsysroot image.

Expanding images

It is convenient sometimes to expand the last partition of an image to fit a larger size. Say you start from a 2GB image, but you need to install an extra amount of software or data that would not fit.

Or, you need to publish an image that takes all the available space on a 32GB memory card.

In order to do that, you specify a `qcow_size` with the total desired size of the image, for example `"qcow_size": "3G"`. The profile needs to be unmounted and renewed upfront, for this to work. Below is an example:

```
$ xsysroot --renew
$ xsysroot --umount
$ xsysroot --expand
Connecting image /home/sysop/scratch/haw.qcow to find and expand last partition
Partition number: 2 start: 40.9MB end: 239MB size: 198MB type: ext3
/dev/nbd12p2: 11/48384 files (0.0% non-contiguous), 11772/193536 blocks
resize2fs 1.42.5 (29-Jul-2012)
Resizing the filesystem on /dev/nbd12p2 to 472064 (1k) blocks.
The filesystem on /dev/nbd12p2 is now 472064 blocks long.

Image partition expanded successfully, new layout:
Partition number: 2 start: 40.9MB end: 524MB size: 483MB type: ext3
/dev/nbd12 disconnected
```

Expansion is applied on the last partition of the image, and only on filesystems of types ext2, ext3 and ext4.

After the image has been expanded, simply mount it again and you should have the extra space available. Consult the specifications of your SDcard to find the exact boundaries of the real physical size.

Xsysroot on a multiuser server

The configuration file `xsysroot.conf` can be placed under the `/etc` directory, or on the user's home directory. If it is found under `/etc`, the users home directory file will not be read.

The reason behind this is because the NBD device files allocated to access the image need to be exclusive for each image. Allowing for multiple configurations would easily end up in a messy combination with unexpected results.

So the best practice to use xsysroot in a multiuser environment is to use one single configuration file located at `/etc/xsysroot.conf`.

This way, all users on the system will have access to all images defined in it.

Additionally, there is a protection mechanism when accessing images from multiple shells. If you try to unmount an image which is currently being accessed elsewhere, xsysroot will complain and tell you which processes are currently running on the image.

You can also find such processes by executing `xsysroot --running` command.

Cooperative distributed work

When several remote users need to work on an image, without access to a central xsysroot server, it can become complicated to distribute the changes across the network, as the images are normally large in size.

Because xsysroot will always work with qcow incremental changes, there exists the option to share the work by sending the qcow image instead, which will normally be much smaller in size.

In this scenario, all users have xsysroot installed on their local systems, with the same configuration and backing images, but they just need to share the qcow image, effectively looking only at the incremental changes.

For example, for a user to receive an upgraded image from another user, he would do the following steps on his local system:

```
$ xsysroot --umount
$ cp updated_qcow_image.qcow $(xsysroot --query qcow_image)
$ xsysroot --mount
```

You should now have the incremental changes automatically applied at your end. Keep in mind that the paths to the backing and qcow image need to be the same, otherwise the backing image path pointers will not be bound correctly.

Building Debian packages

One of the most practical uses of xsysroot is to build software for specific ARM based distributions without the need of the target hardware.

In this chapter we are going to create a very simple project and build a Debian Package natively for the Beaglebone board, in its official Debian Jessie distribution. The project will be a simple print of "hello world".

So let's downlad the latest image, and create a xsysroot profile to access it. Refer to the Tutorial chapter on how to do that.

Chances are that we want to save our project in a source repository, from within the host, therefore we need a way to access the sources from both the host and the guest. We will use xsysrooot `tmp` profile variable for that. Let's create a simple "hello world" app.

```
$ cd $(xsysroot --query tmp)
$ mkdir myapp; cd myapp
```

Create a file myapp.c containing this code snippet:

```
#include <stdio.h>
int main(void) {
  printf ("hello xsysroot\n");
  return 0;
}
```

We are now asking xsysroot to create us a Debian package skeleton folder, and telling Debian builder what to compile and package:

```
$ xsysroot --skeleton .
$ echo "hello /usr/bin" > debian/install
```

Edit a file called `Makefile` containing these rules:

```
all: hello
hello: hello.c
        gcc -Wall hello.c -o hello
```

We are ready to build the package natively from the guest, the Jessie for the Beaglebone system. The `debian` folder contains all the build files, you might want to change those to your package needs.

```
$ xsysroot --chroot
Starting sysroot shell into: /tmp/beaglebone as the superuser
$ cd /tmp/myapp
$ apt-get install devscripts
$ debuild -b -us -uc
```

That's it! At the `/tmp` directory you should have a debian package ready to install on the Beaglebone.

Cross building Debian packages

The QEmu emulator is quite slow to run, so for larger projects you might consider cross building the software. This implies installing a cross toolchain on the host, and running `debuild` on the host instead.

Xsysroot gives you an option to install the build dependencies on the guest image for you, as well as a simple wrapper to call `debuild`. You will need to adapt `debian/rules` to explain how to cross compile.

From within your project's root directory, execute these commands from the host:

```
$ xsysroot --depends
$ xsysroot --build
```

Looking after your images

```
$ xsysroot --list
$ xsysroot --integrity
$ xsysroot --zerofree
```

Python bindings

Xsysroot is written in Python, as one single code module. To expose xsysroot to Python, do a `xsysroot --upgrade`, and at this point you should be able to use it directly:

```
$ import xsysroot
$ print xsysroot.__version__
1.913
```

Accessing xsysroot from Python allows for automating a lot of tasks on foreign images.

As an example, This xsysroot Python script builds the Love 2D framework inside a RaspberryPI image.

Virtualized xsysroot

Xsysroot on a VPS

References